Executive Summary

Automated Testing is a core activity of any agile development methodology. As we move towards continuous deployment, test automation becomes ever more important due to the quick feedback response that it provides to the development team about the health of the application.

In order to get this quick feedback, automated tests need to be executed continuously, should be fast and test results should be consistent and reliable.

In order to achieve these, the majority of the verifications should be done as part of the development of new features. In other words, development and testing should be a coherent activity, and quality should be “baked in” right from the start by ensuring that what is being developed works and that it hasn’t broken existing functionality.

This requires “inverting the test automation pyramid” by pushing down GUI tests that take a long time to execute, to lower levels e.g. API layer that can run straight after unit tests as part of the build to provide the initial level of confidence.

Related:

- **How to choose which tests to automate?**
- **Where to start test automation for existing website**
- **Should Test Automation be Done by Separate Dedicated Team?**

Test Automation Strategy Overview

Prevention rather than detection – while every effort should be spent on preventing the introduction of defects in the application in the first place, the techniques and methods for that are outside of the scope of this post. Here, the methodologies are defined to allow for quick detection of bugs when they are introduced into the system and feedback to development.

Quality should be favored over quantity. In most cases, it is better to release with one feature that is rock solid rather than multiple features that are flaky. As a minimum release criterion, any newly developed feature should not have introduced any **regression defects**.

As already mentioned, quick feedback on the health of the application is of huge importance to support continuous delivery, therefore, a process and a mechanism by which we can obtain feedback quickly is formulated.

One way of getting quick feedback is by increasing the number of unit tests, integration tests, and API tests. These low-level tests will provide a safety net to ensure the code is working as intended and helps prevent defects escaping in other layers of testing.

Unit Tests form the foundations for test automation at higher levels.

The second element of improvement is running the regression tests more frequently and aligned with the process of Continuous Integration, see later. Automation Testing should not be seen as an isolated task, but rather as a coherent activity embedded in the **SDLC**.

Definition of Regression Packs

Automated regression tests are the core of the Test Automation Strategy.

Smoke Regression Pack, which is a sanity check that the application can be loaded and accessed. Also, just a few key scenarios should also be run to make sure application is still functional.

The aim of the smoke test pack is to catch the most obvious issues, such as application not loading, or a common user flow cannot be executed; for this reason, the smoke tests should last no longer than **5 minutes** to give quick feedback in case something major is not working.

The smoke test pack runs on every deploy and can be a mixture of API and/or GUI tests.

Functional Regression Packs, which is meant to check the functionality of the application in more detail than the smoke test.

Multiple regression packs shall exist for different purposes. If there are multiple teams working on different sections of the application, then ideally there should be different regression packs that can be focused on the area the team is working on.

These packs should be able to run in any environment as and when required, provided the behavior of the features remain consistent throughout the environments. They are executed multiple times a day and should last no longer than 15 to 30 minutes.

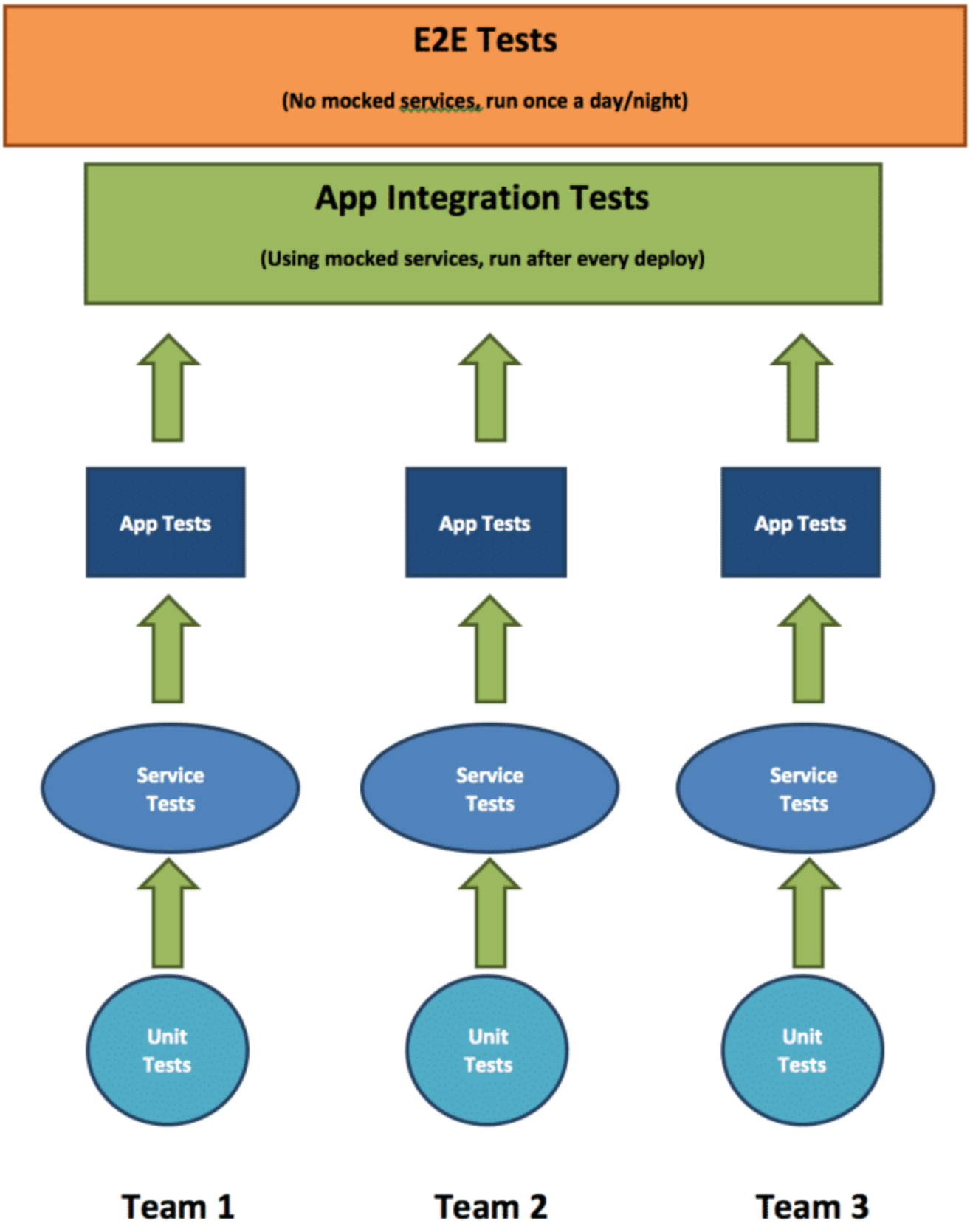
As these functional tests are more detailed, then they will take longer to run therefore, it is important to have the majority of functional tests at API layer where tests can be executed faster so we could be within the **15 to 30 minutes** time limit.

End-to-End Regression Pack, which tests the whole application as a whole. The aim of these tests is to ensure that various parts of the application which connect to various databases and third-party applications work properly.

The End-to-End tests are not meant to test all of the functionalities as those are already tested in the functional regression packs, however, these tests are “light-weight” which just check the transitions from one state to another and a handful of the most important scenarios or user journeys.

These tests are mainly executed through the GUI, as they are checking how users would use the system. The time taken to execute these can vary from one application to another but they are usually run once a day or night.

Test Automation Strategy for Multiple Agile Teams


E2E Tests
(No mocked services, run once a day/night)
App Integration Tests
(Using mocked services, run after every deploy)
App Tests
App Tests
App Tests
Service Tests
Service Tests
Service Tests
Unit Tests
Unit Tests
Unit Tests
Team 1
Team 2
Team 3

Automated Unit Tests

Test Automation starts at the unit level. Unit tests should be written by developers for any new feature that is developed. These Unit Tests form the foundation of a larger automation practice that spans all the way up to the System GUI Tests.

It is the responsibility of the developers to ensure that for every new feature that is developed, a set of coherent and solid Unit Tests are written to prove that the code works as intended and meets the requirements.

Unit Tests provide the most ROI to the team as they are very quick to run, easy to maintain and modify (as there are no dependencies) and when there are errors in code, it is quickly fed back to the developer.

Unit tests are run on the developer's machine as well as the CI environment.

Automated Integration / API or Service Tests

While Unit Tests are based on testing the functions within a class, Integration Tests form the next level up from Unit Tests to test the classes that collectively make up the component to deliver a piece of functionality. These tests are executed only when the Unit Tests have run and passed.

Service Tests are naturally run at API layer without the intervention of the GUI web interface; hence tests would be able to verify functionality in a pure form and because the tests talk directly to the components, they are fast to execute and will be part of the build.

Where necessary, mocks such as **wiremock** will be used to factor out the dependence of other 3rd party systems and when the downstream systems are not available to provide the data required for testing.

Integration Tests and/or Service Tests can be run on the developer's machine as well and be part of the build, but if they start to take a long time, then it is best to run on the CI environment.

Tools such as SoapUI can be used for Service Tests.

Application Testing

A typical e-commerce application can be split into different applications or "apps" that provide different functionalities. The concept of "App Testing" is where a group of tests

that test the functionality of an App are organized together and run against the desired App. This pack will be useful in cases when a team wishes to release an individual App and would like to know whether it is functioning correctly.

Application Tests typically require an interface to interact with the different components, therefore it is anticipated that these tests are run via a browser on the GUI.

The purpose of App Testing is to ensure that features of the application are functionally correct. Because the tests are organized in a manner to provide confidence in the health of a particular app, these tests are normally referred to as Vertical Tests, since they execute “down” a particular app. The tests are very thorough and coverage is large.

Selenium WebDriver could be used to run these automated tests against the browser. This tool is the most popular for browser automation tests and provides a rich API to allow for complex verifications.

End-to-End Scenario Tests

The GUI automated tests which are run against the system, serve as typical user flows, journeys or end-to-end scenarios. Due to issues with this type of tests (discussed below), these will be kept to a minimum. The end-to-end scenarios are included in the nightly regression pack.

Inverting the Test Automation Pyramid

As part of the Test Automation Strategy, we need to ensure to minimize the number of automated tests that are run at GUI layer.

Whilst running automated tests through the GUI provides good and meaningful tests in terms of simulating a user’s interaction with the application, it is prone to many issues as listed below:

Brittle – Because the tests rely on the HTML locators to identify web elements to interact with, as soon as an id is changed the tests fail, therefore they bear a lot of maintainability costs.

Limited Testing – The GUI could confine the tester’s ability to fully verify a feature as the GUI may not contain all the details from the web response to allow verification.

Slow – Because tests are executed through the GUI, the page load times can substantially increase the overall testing time and as such the feedback to the developers is relatively slow.

Least ROI – Because of the above-mentioned issues, the GUI automated tests provide the least ROI.

The Browser Automation Tests will be kept to a minimum and will be used to simulate a user's behavior incorporating common user flows and end-to-end scenarios where the system as a whole is exercised.